	Type	Hits	Search Text
1	BRS	160	object adj oriented and data adj group
2	BRS	11	object adj oriented and data adj group and deposit
3	BRS	3	(717/108.ccls.) and beginning adj2 (access or invo$)
4	BRS	1	(717/116.ccls.) and beginning adj2 (access or invo$)
5	BRS	77	(717/108,116.ccls.) and deriv$ near2 object
6	BRS	54	(717/108,116.ccls.) and deriv$ near2 object and attributes
7	BRS	1	5822587.pn.
8	IS&R	7	(("5161225") or ("5247669") or ("5313636") or ("5487141") or ("5499365") or ("5537630") or ("5560014")).PN.
9	BRS	0	5499365.pn. and thread
10	BRS	12	5822587.uref.
11	BRS	17	facet same subgroup
12	BRS	48	facet adj type
13	BRS	5	facet adj type and object adj oriented
14	BRS	5	facet adj type same object and inherit$
15	BRS	5	facet adj type and object and inherit$
16	BRS	38	facet adj type and object
17	BRS	237	subgroup and object and prototype
18	BRS	44	subgroup and object same prototype

Chapter 8 - Classes, Variables, Methods, and Interfaces-Object-Oriented Programming in Java

In Chapter 1, we observed that we can analyze real-world systems and problems and model them as a collection of objects. We define general classes of objects, gradually add more specific subclasses that inherit their characteristics, and add specific characteristics of their own.

Once we have analyzed a problem or system from an object-oriented point of view and designed a model of that problem or system, an object-oriented programming language gives us a formalized way of expressing that model and its constituent parts.

This chapter will revisit some of the concepts introduced in Chapter 1, examining Java's own implementation of the object-oriented paradigm.

Classes

When you write a Java program, all your code is contained within class definitions. Recall that in object-oriented parlance, a class is a description of a general category of objects that share certain attributes and behavior. In Java, we refer to these attributes as variables, and the behavior as methods. All variables and methods belonging to a class are declared within the definition of that class. Note that this adheres to the concept of encapsulation discussed in Chapter 1. Here is the general form for the definition of a class:

```
class declaration {
   variable declarations
   method declarations
}
```

Actually, Java allows you to intersperse variable declarations with method declarations, even if the methods make forward references to variables declared later in the class. It is better style, however, to place all the variable declarations first and to follow them with the method declarations, as it makes your code much easier to read.

Parts of a Class Declaration

The class declaration denotes the identity of the class. It proclaims not only the name of that class, but also its accessibility and the name of the class that it extends, that is, the class from which it inherits characteristics. (parameters within brackets ([ ]) are optional.)

```
[classModifiers] class ClassName [extends SuperClass]
                 [implements Interface1, ... ,InterfaceN] {
    // class body (variable and method declarations).
}
```

classModifiers

One or more class modifiers specify the accessibility of the class, as well as information about to what extent the variables and methods defined in this class will be visible in classes derived from it. The class modifiers may be omitted entirely, in which case objects of this class can only be declared and accessed by other classes in the same package, and this class may also be subclassed within this package. There are three possible class modifiers: public, abstract, and final.

public
If a class is declared as public, then this class has global accessibility. This means that any class defined in any package may declare and access objects of this class. If a class is not explicitly declared as public, then accessibility is limited to within this package.

A class declared as public must be the only public class declared in its source file. Furthermore, the name of the source file must be the same as the name of the public class, with the .java extension added. For example, a class declared as:

```
public class MyClass { ... }
```

must live in a source file called MyClass.java, and it must be the only public class declared within that source file.

You declare a class as public when you want that class to be globally accessible. In general, it's a good idea to declare your applets as public so that anyone who comes across your Web pages can run your applets. In addition, you may declare packages of related classes, which are intended to provide some general functionality, very likely for reuse by many different programs. In order for these classes to be accessible from outside that package, you must declare them as public. Note, however, that not all of the classes in such a package may be intended for use from the outside; some may be intended only for internal use. You would declare those classes with the default access (i.e., you would leave off the public keyword), and they would thus be invisible outside the package. Notice how this conforms to the concept of data-hiding, described in Chapter 1, which is central to the object-oriented approach.

abstract
An abstract class is a class that contains one or more abstract methods. This may sound like a circular definition, but the following explanation should clear things up.

Sometimes you declare a class for which you want to declare a certain method, but you really want the internal details of that method's implementation to be different for each subclass. For example, consider the class java.awt.Image. It makes sense to declare a class for displayable bitmapped images that is independent of any particular file format—GIF, JPEG, or some as-yet-undefined standard. There are certain things you need to do with an image regardless of the file format, such as get its width or height. Clearly, however, the implementation of these operations is going to differ for each image format.

In cases like these, it is clear that you need to defer the definition of such methods to subclasses. To do this, you simply leave the definition blank and declare it as an abstract method. If a class contains one or more abstract methods, then the class itself must be declared as abstract or the compiler will flag it as an error.

You cannot directly instantiate an object of an abstract class. In order to use an object with the characteristics described in a given abstract class, you must create a subclass that includes definitions for the methods left blank in the abstract class. You can then instantiate objects of the subclass.

abstract classes provide Java with what object-oriented programmers call pure polymorphism. In pure polymorphism, there are classes (abstract classes) that cannot themselves be instantiated, whose purpose is to serve as a base class for a group of related classes. These base classes are so generic that it is hard, or even impossible, to define the details of their behavior. The details are left to the subclasses, and each

subclass may implement these behaviors differently. Thus, we start with no implementation, and we end up with multiple distinct implementations. Hence, the term pure polymorphism.

Like any other classes, abstract classes may be declared as public or with the default access, in which case they will only be visible within the package.

final
Classes declared as final may not be subclassed. You may have noticed that this is, in a sense, the opposite of an abstract class. The compiler will notice it, too; you may not declare a class as both abstract and final. If you make such a declaration, it will be flagged as an error by the compiler.

At first glance, it is not obvious why you might want to declare a class as final, thereby limiting its ability to take advantage of inheritance, one of the most crucial features of object-oriented programming. There are two very good reasons you might want to do this. One concerns efficiency; as we shall see, it is possible for subclasses to override the methods defined in their parent classes. Thus, when a method of a given class is called, it is not possible to determine at compile time which version of that method will be executed: the version of the method defined in that class or an overriding method of a subclass with the same name and parameter types. This must be determined dynamically at run-time, each time that the section of code is run, depending on tables generated when the object is instantiated. Each time this determination is made, it is conceivable that it could require the loading of a class not previously resident in memory. There may be a performance cost associated with loading the new class. A final class, however, may not be subclassed. Because this guarantees that its methods will not be overridden, the instructions comprising the method may be inlined into the compiled code, eliminating the need to make a decision and possibly load a new class while the program is running.

The second reason for declaring a class as final concerns both security and quality assurance. If you have specified that a class has certain behavior, you may not want to allow anyone to modify that behavior by subclassing it and overriding its methods. Declaring the class as final prevents all subclassing and allows you to guarantee the behavior of your class with respect to both functionality and security.

ClassName
A class name is an identifier, and as such it must obey the naming rules specified in Chapter 6. Class names within the same package must be unique. Additionally, if you import classes from another package, the class names you declare in your package must not conflict with the names in the imported package.

By convention, Java programmers capitalize the first character of a class name, as well as the first character of each new word within the name, for example: ThisClass, ThatClass, and TheOtherClass. This convention, while not required by the language or the compiler, provides the convenience of distinguishing class names from variable names, which typically begin with lowercase letters. This makes your code easier to read.

extends SuperClass
This clause specifies the class from which this class inherits—that is, its immediate superclass, or alternatively, its parent class. You can think of the subclass as extending the functionality of the parent class, hence the keyword extends. Remember, Java has only single inheritance, so you can only specify one superclass in the extends clause. Recall also that if you have declared a class as final, then it cannot be subclassed—that is, it cannot appear in any class's extends clause.

If you omit the extends clause, your class will inherit by default from a built-in class called Object. Class Object is the root of all classes and does not inherit from any other class.

Inheritance, as we have pointed out, is one of the central concepts of OOP. Like any good object-oriented language, Java gives you the power to control inheritance, that is, to specify, either explicitly or by default, which features of a class will or will not be visible within its subclasses and which methods they may or may not override. By default, a class inherits all fields (variables and methods), except constructors, from its superclass. Other than constructors, all variables and methods are inherited, although as we shall see, those declared as private will be invisible to any classes inheriting them. Although constructors cannot be inherited, we shall see later how a subclass can access its parent's constructors.

implements Interface1, ... , InterfaceN
Although Java has only single inheritance, the designers of the language were sensitive to the fact that there are times when programmers need the functionality provided by multiple inheritance. In recognition of this, they provided the interface mechanism, which provides some of the functionality of multiple inheritance while avoiding the circularity issues and other complexities and dangers that true multiple inheritance incurs. We will discuss interfaces in detail later in this chapter. At this point, the key things to notice are:

• A class can only extend one superclass, but it can implement as many interfaces as you please or none at all. • The interfaces are specified in the class declaration, after the implements keyword, as a comma-separated list. • The implements clause is optional; if your class does not implement any interfaces, you can omit the implements clause entirely.

Variables

Simply put, variables hold data. An object-oriented way of expressing this is to say that variables embody the state of a given object. Generally speaking, variables come in two flavors: there are instance variables, whose data pertains to individual objects of a class, and there are static variables—sometimes called class variables —whose data pertains to all objects of a given class. Each individual object of a given class is called an instance, or an instantiation, of that class. When you create a specific object of a class, you are said to instantiate an object of that class. When you instantiate an object of a class, space is allocated for that object's instance variables. Each object that you instantiate has space allocated independently for its instance variables.

static variables, on the other hand, are created only once and are shared by all objects of a given class and any inheriting subclasses. No matter how many objects of that class you create, there will be only one copy of the static variables belonging to that class. For example, in the class java.lang.Integer, there is a variable MAX_VALUE, which is declared as static. This variable reflects the largest value an Integer is allowed to have. Because this number is the same no matter how many Integer objects you instantiate, it makes sense to allocate memory for only one copy of this variable, and that is in fact what is done.

To refer to the instance variables of an object, you use the name of the specific instantiation, followed by a period, followed by the name of the instance variable. For example, suppose you have a class called StarShip, one of whose instance variables is named powerRemaining. You have instantiated a StarShip object called excalibur and another called meridien. You can refer to the powerRemaining of these instances as excalibur.powerRemaining and meridien.power-Remaining, respectively.

To refer to static variables, you may either refer to them by the class name, followed by a period, followed by the variable name or you can substitute the name of any instance of that class for the class name; the result will be the same. For example, if we have declared in class Starship a static variable called maxSpeed, then we may refer to it as StarShip.maxSpeed, or excalibur.maxSpeed, or meridien.maxSpeed. We will get the same value, and any changes we make will affect all objects of class StarShip.

Within a class definition, instance variables and static variables belonging to that class can be referred to without qualifying them. For example, within the class Starship, we can refer simply to powerRemaining or maxSpeed. As we shall see, however, we cannot make such references to instance variables from within static methods.

The Three Special Variables: this, super, and null

Each object has three built-in variables, represented by the keywords this, super, and null. this is an alias to the object itself. Because the code you write within a class is written from the point of view of a single object (except static methods, as we shall see) and because there may be many objects of that class instantiated with different names, you need to have a name by which you can refer, within the non-static methods of a class, to the current object itself—that is, the object currently performing this method.

Within a class definition, you may refer to the instance variables of that class without specifying an object. For example, within the declaration of the class StarShip, we can simply refer to powerRemaining, although there is an implicit this. in front of the name of the variable (this.power-Remaining). For the most part, you don't need to explicitly type the this. However, there are certain circumstances when you need to use this explicitly.

Consider the example below. this.n is an instance variable of the class, while no dot is a local variable of anyMethod(). The explicit this is necessary to distinguish between the local variable i and instance variable this.i.

super is an alias to an object's immediate superclass (the class named in the extends clause for this class). Typically, because variables in the superclass are inherited by the subclass (unless rendered inaccessible by the keyword private), you simply refer to those variables without qualification as instance variables of the local class. However, it is possible to declare an instance variable in a subclass with the same name as an instance variable in its superclass. The variable defined in the subclass will then mask or "shadow" the variable defined in the superclass. If you want to refer explicitly to the variable defined in the superclass, you use the keyword super, as shown in example below.

```
class SupCls  {
  int i=10;  // i is initialized on instantiation
}
class Cls extends SupCls  {
   int i, j;
      public void anyMethod (int i)  {
         this.i = i;    // explicit "this" is needed because there
                        // is a conflict on name.
         j = super.i;   // j = 10
      }
}
```

The keywords this and super can only be used within a non-static method. When we discuss static

methods later in this chapter, it will become clear why this is the case.

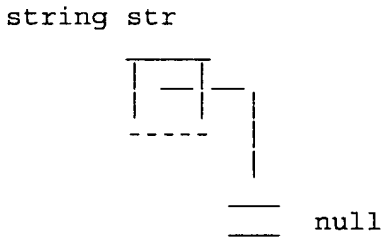
When you first declare an object, it has no value until it is initialized. Before initialization, the object has the default value of null, which essentially means, "no value." There are times when you will want to explicitly assign the value of null to an object—for example, to terminate dynamic data structures such as trees and lists. Structures like these may grow or shrink in size, with no preset bounds. You use null-valued objects as markers for the endpoints of these structures. Another case in which it is useful to assign null to an object is the case of a Thread. To stop a Thread in Java, you can set its value to null.

You can assign the value null to objects of any class but not to methods or variables of primitive data types, such as ints, floats or chars.

Consider the declaration:

```
String str;
```

After this declaration, str is considered to be the handle of String object. The concept of a handle is similar to the idea of a pointer in C or C++. Unlike a pointer, however, a handle is not a physical address. After the preceding declaration, there is no memory allocation to str, and thus str is considered a null handle. Graphically, we represent a null handle using a notation borrowed from electrical circuit design, the symbol for the electrical ground:



When we use the word handle, we are not using it in the sense that it is used by Macintosh and Windows C or C++ programmers. This is not a "pointer to a pointer," as the word handle often implies on those platforms. In Java, a handle is merely a reference to an object. Again, there is no implied physical address.

Parts of a Variable Declaration

The general form of a variable declaration is:

```
[variableModifiers] type variableName [= initialValue];
```

Variable Modifiers

Variable modifiers specify the accessibility of variables declared at the class level. You do not need to specify variable modifiers if you intend for a given variable to have the default accessibility. By default,

a variable defined in a given class will be:

• inherited by and visible within any classes declared within the package that extend this class • accessible to all other classes within the package
• inaccessible outside the package

If you intend otherwise, you can indicate this by specify one of the following: public, protected, private, static, or final.

public
A variable declared as public has the same accessibility as its class. Intuitively, it is clear that a variable's accessibility cannot be wider than that of the class in which it is declared. (Thus, in a class with default access, a public variable will be accessible only within the package, but in a class with public access, a public variable will be accessible everywhere.) Variables declared as public are inherited by any subclasses derived from the class in which they are declared.

protected
Variables declared as protected are accessible by all classes within the package containing the class in which they are declared. However, outside the package, protected variables can only be accessed by subclasses of the class in which they are declared. In other words, protected variables are relieved of the default restrictions on inheritance, but they retain restrictions with regard to access by unrelated classes outside the package.

This is not the same thing as a protected variable in C++. In C++, protected variables are only accessible in a given class and its subclasses. Java adds the concept of the package, inside which a protected variable is accessible to all classes, even those not derived from the one in which the variable is declared.

---

private
private variables can only be accessed within the class in which they are declared. This restriction extends to derived classes as well. Even in a subclass of the class in which they are declared, private variables are inaccessible.

static
Variables declared as static are instantiated only once and shared by all objects of the class in which they are declared and by any inheriting subclasses. Note that there is an important distinction between static variables, which belong to the class as a whole, and instance variables, which are separately created for each object of the class that is instantiated and thus "belong" to that object. All variables not explicitly declared as static or final are instance variables.

The static storage class specifier may be combined with any of the access specifiers: public, protected, private, or the default.

final
Variables declared as final are, for all intents and purposes, constants. When a variable is declared as final, a value must be assigned to that variable immediately, in the declaration. Thereafter, the value of the so-called variable cannot be changed. It is often to declare a constant variable as final static.

In addition to the modifiers we have described, the Java specification reserves the keywords transient and volatile. However, these two storage class specifiers are not yet implemented in version 1.0.

---

type
This simply states the type of this variable, which may be any primitive type or any array or object type, including an object of this class. Types were discussed in detail in Chapter 6.

variableName
The rules for naming variables were discussed in Chapter 6. As a matter of style, it is best to choose a variable name that starts with a lowercase letter, such as counter, smallestValue, or myOtherDouble.

[= initialValue]
As we have pointed out, a variable can optionally be initialized on the same line on which it is declared. It is a good idea to do this whenever possible, because it is an error to try to access the value of a variable that has not yet been initialized. You can initialize a variable to the value of another already-initialized variable or any expression that evaluates to the appropriate type. For example:

```
int myInt = 6;
int yourInt = myInt + 14;
myObject m = new myObject();
int[] foo = {1, 2, 3, 4, 5};
```

Methods

All executable statements in Java are contained within methods (for an exception, see the sidebar, "Static Blocks"). All methods reside inside a class. This is a stricter approach to object-orientedness than that taken by C++, where functions may exist independently from objects if the programmer so chooses.

Static Blocks *(Sidebar)*

There is one exception to the rule that all executable code must reside within methods. Java programmers may use static blocks to specify some code that must occur as a given class is instantiated. Typically, this code is used to initialize static variables that may not be able to be fully initialized on their declaration line, such as arrays.

The static block begins with the keyword static, after which it is enclosed by curly braces—{}. Inside the static block, only static variables may be referenced, although variables local to the block may be declared.

Consider the following example:

```
class Foo {
   static double[] theCosines = new double[100];
   static {      // this is the static block
      for (int i = 0; i < 100; i++) {
         theCosines[i] = Math.cos((double) i / 100.0);
      }
   }
   ...
}
```

Notice that the static block only references static variables and methods. Even the local variable i is implicitly static because it is declared locally within a static block. (Because it is declared within that block, however, it is not accessible outside the block.)

This static block is necessary if we expect there to be specific values in the array Foo.theCosines without necessarily instantiating any specific objects of class Foo.

Argument Passing

When you call a method, you send it information by passing that information as a parameter (also called an argument) to that method. When the method is called, whatever is passed to these parameters is substituted for them inside the method's execution.

Parameters of primitive types (such as int, float, char, and boolean) are passed by value. This means that a local copy is created with the same value as the actual parameter; if you modify the parameter within the method, you only modify the local copy, and the change has no effect on the state of the actual parameter, outside of the method itself.

Object and array types, on the other hand, are passed by reference. This means that inside the method, when you refer to that parameter, you are referring to the actual object or array that was passed in—there is no local copy, as there would be in a call by value. The result of this is that if you make changes to the state of an object passed in as a reference parameter, then the changes you make will affect the actual object, and therefore those changes will persist even after your method exits.

When you declare a method, you state explicitly what types of parameters it expects to receive. This is done through the method's argument list, the syntax of which will be discussed soon, when we look in detail at the parts of a method declaration.

Note that in object-oriented programming, you don't have to pass a method all the information it requires through its parameter list. This is because the method already knows about—and has access to—all the instance variables and static variables of the class in which it is defined. The only items that need to be specified in the parameter list are those to which the method does not already have access, such as the instance variables of other objects or those that may be different each time the method is called.

The following example illustrates several issues of relevance to parameter passing:

```
class Foobar {
   double d = 3.14;
   public void fooMethod (double x, Foobar other) {
       x = x * 2.0;    // no effect outside this method
       d = x * other.d;
       other.d = 0.0;    // has global effects
   }
}
public class MainClass {
    public static void main(String args[]) {
        double ourDouble = 28.8;
        Foobar ourFoobar = new Foobar();
```

```
        Foobar ourOtherFoobar = new Foobar();
        ourFoobar.d = 14.4;
        ourOtherFoobar.d = 19.2;
        System.out.println("Before the call to fooMethod(...)");
        System.out.println("  ourDouble is " + ourDouble);
        System.out.println("  ourFoobar.d is " + ourFoobar.d);
        System.out.println(" ourOtherFoobar.d is " + ourOtherFoobar.d);
        System.out.println("");
// Here's the method call of interest...
    ourFoobar.fooMethod(ourDouble, ourOtherFoobar);
    System.out.println("After the call to fooMethod(...)");
    System.out.println("   ourDouble is " + ourDouble);
    System.out.println("   ourFoobar.d is " + ourFoobar.d);
    System.out.println(" ourOtherFoobar.d is " + ourOtherFoobar.d);
   }
}
```

The output of this program will be:

```
Before the call to fooMethod(...)
   ourDouble is 28.8
   ourFoobar.d is 14.4
   ourOtherFoobar.d is 19.2
After the call to fooMethod(...)
   ourDouble is 28.8
   ourFoobar.d is 1105.92
   ourOtherFoobar.d is 0
```

When we call ourFoobar.fooMethod(...), we don't need to pass it a value for d. The object ourFoobar already knows about d, because d is an instance variable of the class Foobar, and any instantiation of a Foobar knows the value of its own d. What it doesn't know is what number we want to pass to x in its fooMethod(...). Nor does ourFoobar know anything about the values of instance variables in ourOtherFoobar, such as ourOtherFoobar.d. This is why we have to pass these items through the argument list of fooMethod(...).

Notice that the changes made to x within the method do not have an effect on ourDouble, because ourDouble is a primitive type and is passed by value. The changes we make to other.d, however, have global effects on the state of the object ourOtherFoobar, because, being an object, it is passed by reference.

Overloading Methods

The signature of a method is defined by its argument list—specifically, by the number, order, and type of each argument. Methods are distinguished by the combination of their name and their signature. In Java, as in C++, it is possible to create overloaded methods. These are methods that have the same name but different signatures. It is often useful to provide similar functionality for many different types of arguments and to give the methods that provide that functionality the same name. When the methods are called, there is no ambiguity as to which method will be executed, because methods with the same name are distinguished from one another by their signatures.

For example, in the class java.lang.String, the following two methods are defined:

```
public static String valueOf(int i){ ... }
public static String valueOf(float f){ ... }
```

The method String.valueOf(...) is intended to take whatever primitive data type is passed to it as a parameter and convert that parameter to a String representing its value (for example, from the floating-point number 2.4 to the three-character String "2.4"). It's clear that one needs to go through different steps to make a String out of an int than to make a String out of a float. This is where method overloading comes in handy. If you call this function with an int as a parameter, it will use the first version listed above; if you call it with a float, it will use the second version. The compiler uses the signature to determine which version to execute.

Earlier, we said that abstract methods allow Java to provide pure polymorphism. Method overloading is the mechanism by which Java provides another type of polymorphism, ad hoc polymorphism., in which the compiler decides which version of a method to call based on the signature with which it is called. Contrast this with the pure polymorphism provided by abstract types, where the decision as to which version of a function to use may often need to be deferred until run time.

Overriding Methods

Often, it is useful to have a subclass that defines its own version of a given method, with the same name and signature as that in its superclass. This is called method overriding. The new method in the subclass is called the overriding method, and the method in the superclass is said to be overridden.

Inheritance is useful because it allows the programmer to reuse existing code in defining new classes. Method overriding provides inheritance with needed flexibility. Because of method overriding, you can keep only those methods of the superclass that are appropriate in the subclass and provide a new version of any whose implementation you wish to alter. abstract classes rely on method overriding in order to provide pure polymorphism. Not only abstract methods may be overridden, however; any non-static, non-final method in a non-static, non-final class may be overridden when the class is subclassed.

Constructors

As we have noted, mere declaration of an object does not allocate space for the object's instance variables. When an object is declared, it has a value of null until it is instantiated. It is an error to attempt to access the instance variables of an object before it has been initialized. You instantiate an object with the new operator. This can be done on the same line as the declaration, for example:

```
Planet p = new Planet();
```

If the use of Planet() on the right-hand side of the assignment looks like a method call, that's because it is. It is a call to a special method defined for every class, called a constructor.

A constructor is a special method that shares the same name as the class in which it is defined. Its job is to initialize the instance variables of an object when the object is first instantiated. Although in most respects a constructor is like any other method you define, there are certain things that distinguish constructors from other methods:

1. **Constructors have no return types.** A constructor has an implicit return type of void. Hence, no return statement is required in the body of a constructor.
2. **Constructors are not inherited by subclasses.** Each subclass uses its own constructor, whose name is the same as the class. Although constructors are not inherited, it is possible on the first line of a constructor to explicitly call the constructor of your superclass, by using the super keyword as a

constructor call. We'll discuss this shortly. **3. Constructors are invoked differently than ordinary methods.** When you invoke an ordinary method, you qualify it with the name of the object to which you are sending the message to perform that method; for example:

```
str.substring(5);
```

A constructor invocation, on the other hand, needs no qualification. You simply invoke the constructor with the constructor name (i.e., the name of the class), typically preceded by the new operator. The new operator indicates that some new storage is being allocated. A constructor is invoked by the constructor name (i.e., the class name), usually after the new operator:

```
String str = new String("I am a string.");
```

4. A constructor may not throw an exception. If a constructor calls a method that could throw an exception, it must be caught and handled within the constructor. We will cover exception handling in Chapter 9.

In other ways, constructors are like ordinary methods. One very useful implication of this is that, like ordinary methods, constructors can be overloaded. Like ordinary overloaded methods, overloaded constructors are distinguished by the signature embodied by their argument lists. This allows you to provide flexibility in constructing objects in your class. For example, the class java.lang.String provides no less than seven overloaded constructors. Here are a few of them:

```
String()          If given no parameters, the String() constructor
                  constructs an empty string.
String(String)    Given a String as a parameter, this will construct a new
                  String, which is a copy of the one passed in.
                  (A constructor like this is called a copy constructor,
                  and many classes use them.)
String(char[])    This constructs a new String from an array of characters.
```

this() as a Constructor

Because constructors can be overloaded, it is often convenient within a constructor to call another constructor for the same class. It is a good idea, for example, to provide a default constructor for your class; that is, a constructor with no arguments. Imagine you have already defined a constructor MyClass(int i, int j), and you want the default case to be equivalent to MyClass(0, 0). There is no need to duplicate the code you already provided in MyClass(int i, int j). Instead, you would define your default constructor as follows:

```
MyClass () {          // default constructor -- no arguments
    this(0, 0);       // must be first statement
}
```

The call to this(0, 0) invokes the constructor of the same class that has the signature (int, int). The only place you can use the this keyword as a constructor call is within a constructor, and it must be the first statement of that constructor; otherwise it is an error, and the compiler will catch it.

super() as a Constructor

When you declare a derived class, you inherit instance variables from the superclass. Because the constructor for the derived class is not itself inherited, you need a way to initialize the inherited instance variables. You could initialize the inherited variables manually in your constructor—there are times when you may want to do this—but often, it is more convenient and appropriate to let the superclass take care

of initializing these variables, especially if you have inherited private variables to which you do not have access. Fortunately, you are able to call the constructor for the superclass using the super keyword as a constructor call. Just as with the this() constructor, the super() constructor, if used, must appear as the first statement in the body of your constructor method:

```
class MyClass extends MySuperClass {
   double myDouble;
   MyClass(int i, int j, double d) {
      super(i, j);  // calls MySuperClass(i, j);
      myDouble = d; // wasn't initialized by super(...);
   }
   ...
}
```

You can declare a class without specifying a constructor. If your class inherits directly from another class and does not declare any instance variables of its own, then this is the appropriate thing to do. If you omit the constructor from your class, the Java compiler automatically generates a default constructor for that class, which simply calls the default constructor for the superclass:

```
class FinalFrontier extends Space {
   // no constructor supplied
   // compiler automatically generates:
  FinalFrontier() {
      super();  // calls Space();
  }
  ...
}
```

If, however, the immediate superclass does not have a constructor that takes no arguments, then a compiler error will result.

private Constructors

You can declare a constructor as private and then provide a public static method that calls the constructor and returns the constructed object. To create the object from outside the package, you call the static method. This is useful for checking the arguments for the constructor and allowing an exception to be thrown if they are incorrect. Remember, constructors themselves may not throw exceptions, but the static method described earlier may look at the arguments and, if they are correct, call the constructor; if they are incorrect, then the constructor never gets called, and the static method throws an exception instead. Exception handling is covered in Chapter 9.

Parts of a Method Declaration

The format of a method declaration is as follows:

```
[methodModifiers] resultType methodName ([argumentList])
 [Throws Exception1, ... , ExceptionN]
{
    // Method body
}
```

Method Modifiers

The method modifiers public, protected and private have essentially the same meanings as when they are

used as variable modifiers. There are also several additional modifiers defined for methods, which we will discuss here. If no method modifiers are defined, then the default access is the same as for variables; methods with no modifiers are:

- inherited by, visible within, and able to be overridden by any classes declared within the package that extends this class
- visible to all other classes within the package
- inaccessible outside the package

public
public methods have the maximum accessibility allowed by the class in which they are declared. This means that if the class is declared as public, then any public methods contained therein are accessible everywhere, by every other class, whether in the same package or not. If the class is declared with default access, then public methods are accessible only to other classes within the package. In other words, unless the class is declared as public, declaring its methods as public does not change their access relative to the default.

protected
Again, unless the class is declared as public, the keyword protected does not signify anything other than default access. If, however, the class is declared as public, then protected methods may be accessed by any class within the package and by any derived class, whether in the same package or not. Derived classes, inside or outside the package, may also override these methods. Nonderived classes outside the package, however, may not call these methods. The main reason to declare a method as protected is to remove the default restrictions on the method's inheritance outside the package.

private
Like private variables, private methods are not accessible by classes other than the class in which they are declared, including derived classes.

static
We have seen that some variables are instance variables, and others are static. The same is true of methods. In accordance with the object-oriented notion of encapsulation, instance methods have access to the instance variables of a particular instantiation of an object. Instance methods are invoked using the name of the object as a qualifier, as in the following examples:

```
theObject.itsMethod(anArgument);

yourString = myString.replace('a', 'b');

while (!thisBook.finished()) {
   sleep[tonight++] = 0;
}
```

static methods, on the other hand, like static variables, are instantiated once for all objects of their class. As such, they cannot refer to instance variables, because instance variables belong to specific objects, not to the class as a whole. static methods can, however, access static variables. static methods are implicitly final and cannot be overridden by derived classes.

You can invoke a static method using either the name of the class as a qualifier or the name of any

instantiated object of that class as a qualifier; the result is the same. In the following example, the three calls to String.valueOf(int) produce exactly the same results, because that method is defined as static in java.lang.String:

```
String myString = new String();
String otherString = new String();

myString = String.valueOf(4);              // These...
myString = myString.valueOf(4);            // ...are...
myString = otherString.valueOf(4);         // ...equivalent.
```

We mentioned earlier in this chapter that static methods may not use the keywords this and super. Now, it should be apparent why this is so. static methods are shared by all instantiations of the class and thus do not "belong" to any specific instantiation. Clearly, it makes no sense for such a method to make a reference to this. There could be a thousand or more objects that are instances of the class, all sharing this one static method. If we used the keyword this in a static method, which this are we referring to?

The case is similar with super. The super keyword refers specifically to this instance of this class as an instance of the superclass. static methods do not belong to any specific instance of the class, so the reference makes no sense. Additionally, the meaning of the super keyword presupposes that we know what level of inheritance we are at. Otherwise, what class is the superclass? It makes no sense to distinguish between the superclass and the subclass in a method whose scope spans all levels of inheritance. Thus, in a static method, both this and super are undefined.

A common mistake of a Java beginner is to reference non-static fields in a static method. For example:

```
public class Test {
   String str = "I am a string. ";         // shorthand - constructor
                                           // is called implicitly
      public static void main(String args[]) {
         System.out.println(str);          // Wrong! str is a
                                           // non-static variable.
         System.out.println(this.str);     // This won't work
                                           // either:
                                           // can't use "this" in
                                           // static method
      }
}
```

This mistake is most often made by programmers attempting to convert an applet into an application. We will describe a workaround for this problem in Chapter 11.

Notice that we do not explicitly call the constructor for String in creating str. The Java language, in order to treat strings more like "normal" types, allows this shorthand initialization for strings.

---

final

A final method cannot be overridden by derived classes. The final keyword may be combined with public, private, protected, or static, or it may be specified by itself to combine it with the default access

specifier.

abstract
We introduced the concept of abstract methods early in the chapter, in order to explain abstract classes. The concept of an abstract class and that of an abstract method are intrinsically tied. Recall that an abstract method is declared, but no implementation is deferred to the subclasses. Here is an example of what the declaration for an abstract method looks like:

```
abstract void doThatVoodoo(String theMagicWords);
```

Because abstract methods only make sense in a context where they can be visibly inherited and overridden, it follows naturally that the abstract keyword cannot be combined with any method modifiers that restrict inheritance or overriding; in other words, if a method is declared abstract, it cannot be private, static, or final. By the same reasoning, it makes sense that because constructors are not inherited, they cannot be declared abstract.

native
As hard as we try to write platform-independent code, there are things that can only be done in a language that compiles to native code—instructions for a specific machine. Perhaps we have some interesting hardware device that we need to communicate with, or maybe we need to use a library of C function calls provided by a database package. Perhaps we have a particular method that is very computationally expensive and runs too slowly when interpreted by the virtual machine. We can optimize performance by writing such a method in a language such as C, which compiles to native machine instructions.

A native method is essentially a "callout" to a function written in another programming language. Currently, the only other languages for which Java supports native methods are C and C++, but it is expected that support for other languages will be provided in the future.

synchronized
Because Java is multithreaded, it is important that the language provide some sort of concurrency control. That is, there are some operations that can be performed in parallel without difficulty, but there are other operations—in particular, when multiple threads are involved in reading and writing the same shared data—that require some delicate handling. You declare a method as synchronized when you want a guarantee that only one thread may be executing this method at a given time. We will cover synchronized methods in slighty greater detail in Chapter 10, when we explore threads as part of the Java API.

returnType
All methods, except constructors, must be declared with a return type. If your method does not return any value, you must explicitly declare it with a return type of void.

This is different from the case in C and C++, where a function that omits the return type from a declaration is implicitly declared with a return type of int. In C, it is poor style to omit the return type; in Java, it is an error.

A method may return any type, including object types and primitive types. Unless your method returns void, you must include a return statement in your method body, and it must match the return type specified in the method declaration. Methods returning void may omit the return statement entirely and will automatically return at the close of the method body; alternatively, a return statement may be specified within the method body, at which point execution of the method will cease and control will be returned to the calling method.

The compiler is very good at making sure your method returns a value. Even if you include an appropriate return statement in your method, the compiler traces any if...then...else statements in your method to make sure that there is no branch of control that could lead to the method terminating without a return statement.

```
class ManyHappyReturns {
   void macArthur(String s) {
      if (s.equals("I shall return.")) {
      return;  // early return -- OK
   }
         ... // statements
         return;    // this statement is optional
         }
         int wontCompile(Boolean b, Boolean t) {
         int i;
         double d;
         if (b) {
         ...
         return;  // error: must return int
         } else if (t) {
         ...
         return d;  // error: d is not an int
         } else if (i > 0) {
         return i;  // OK
         } else {
         ...
         // (no return statement in this branch)
         // error: all branches must...
         // ... lead to a return statement ...
         // ... unless return type is void.
      }
   }
}
```

methodName

Method names must follow the naming rules described in Chapter 6. Additionally, as a matter of style, it is a good idea to choose method names that start with lower case letters. This convention helps distinguish ordinary method calls from constructor calls, because constructors will be named after their class, and class names (also by convention) begin with upper-case characters.

```
justStandThere();              // method call, no arguments
doSomething(someArgument);     // method call with arguments
Car myPorsche = new Car(java.awt.Color.black); // constructor
                                               // call
myRealCar = herBug.rebuild(); // calling a method with return
                              //  value
```

argumentList
This is a comma-separated list of the parameters (or arguments) this method expects to be passed. The entire list is enclosed in parentheses—(). The parentheses are required, even if your method takes no arguments. This enables the compiler to distinguish method declarations from variable declarations. Within the argument list, each parameter is preceded by its type. For arrays, you follow the type name with []. (Alternatively, you may use the C style of following the variable name with []. The following example shows both styles, but for consistency, you should choose one style and stick with it.)

```
void doItWith(String s, int[] myIntArray, double myDoubleArray[]) {
   // takes a String, internally called s
   // ... an array of ints, internally called myIntArray
   // ... and an array of doubles, internally called
   // myDoubleArray
}
```

throws Exception1, ... , ExceptionN
Defining the term exception is a little tricky. To define it as an error would be incorrect, because an exception and an error are two different things. We will discuss exceptions in Chapter 8, but for now, let's use the rough definition that an exception is an exceptional, and possibly erroneous, condition that arises during the execution of a program. It is generally expected that an exception will occur relatively infrequently. If your method does anything that might generate an exception, you must either handle that exception within your method or state in your method declaration that you do not handle it—that instead, you have chosen to "throw" that exception. The method that called your method must, in turn, choose whether to handle that exception or throw it, and so on.

If a method does not throw any exceptions, the entire throws clause may be omitted. If a method throws one or more exception types, the exception types are listed after the throws keyword, separated by commas.

Interfaces

An interface is essentially a special kind of abstract class in which you specify a set of constants and abstract methods that can be inherited by any class. A class that is designed to "inherit" an interface is said to implement that interface. Perhaps the most important feature of interfaces is that, whereas a class can only extend one superclass, it can implement any number of interfaces—or none at all. Interfaces thus allow objects to support common behaviors, although their internal implementations may differ greatly. Because they support these common behaviors, interfaces can present themselves as similar and interchangeable to outside entities requesting these behaviors. Their internal differences notwithstanding, their outside appearance with respect to these behaviors is the same; hence, the name interface.

In order to implement a given interface, the implementing class must do two things:

1. include that interface in the implements clause of its own class declaration
2. provide non-abstract definitions for all the methods declared in the interface

All methods declared in an interface are implicitly abstract. Thus, it is not necessary to include the keyword abstract in method declarations within an interface, although it is not an error to do so. Similarly, all "variables" declared in an interface are implicitly static and final—because they are

conceived as constants—and the inclusion of these keywords in variable declarations is not necessary within an interface, although it is not an error. That said, it is not a bad idea to include these keywords in your variable and method declarations; being explicit improves the clarity of your code.

You can declare an interface that extends another interface. It will inherit all of the methods and variables defined in the interface that it extends. Unlike classes, an interface can extend multiple interfaces, inheriting all variables and methods defined therein. Any class that implements such an interface must provide definitions for all of the methods defined in that interface and all the interfaces it extends:

```
interface Simple1 {
   int a;
   void absMethod1();
}
interface Simple2 {
   String str;
   void absMethod2();
}
interface Combined extends Simple1, Simple2{
   void absMethod3();
}
// ImpClass is required to implement absMethod1(), absMethod2(),
// and absMethod3().  Otherwise, ImpClass would also need to
// be declared as an abstract class.
class ImpClass implements Combined {
   public absMethod1() {
      ...  // definition provided here
   }
   public absMethod2() {
      ...  // definition provided here
   }
   public absMethod3() {
      ...  // definition provided here
   }
   // other class methods
   ...
}
```

If a class implements an interface, then its subclasses inherit, and may override, the implemented methods (subject to any restrictions imposed by the access specifiers for that class). Additionally, if a class implements a given interface or set of interfaces, it is not necessary for those interfaces to be explicitly included in the implements clauses of any subclasses.

Interfaces as a Substitute for Multiple Inheritance *(Sidebar)*

Multiple inheritance, often referred to as MI by acronym enthusiasts, is one of the most controversial features of many object-oriented languages. This is the idea that one class may be derived from more than one superclass. Java does not support multiple inheritance. In this sidebar we examine the reasons for this, as well as the alternative solution that Java supplies via interfaces.

Multiple inheritance, while providing a certain amount of convenience and flexibility, is also a source of complexity. Furthermore, unless managed very carefully, it is a potential breeding ground for programmer errors that can be hard to trace and debug.

There are two main categories of errors that can result from misuse of multiple inheritance: ambiguity and circularity. Ambiguity results when two or more ancestors of a class each define a variable or variables with the same name(s). For example, let's say you're writing in C++ and you declare a class Ancestor_1, in which there is a variable x and a method myMethod(), and another class Ancestor_2, which declares its own variable x and its own myMethod(). You then decide to declare a class Descendant that inherits from both of these ancestors. It therefore inherits two variables, both named x, and two methods, both named myMethod(). How can we know which x or which myMethod() we mean when the Descendant class refers to them? There is no need to go into the specific syntax here, but suffice it to say that the ambiguity in such cases is resolved by the programmer having to specify which x or myMethod() is intended whenever a reference to one of them is made in the derived class. This presupposes that the programmer knows there is an ambiguity and is aware of its source. As we shall see, this is not always easy.

Circularity results when a programmer attempts to define a class that inherits, directly or indirectly, from itself. Questions such as "If your ancestor is also your descendant, whose methods override whose?" result in a classic chicken-or-egg game that would destroy the clarity of the inheritance structure.

The problem with multiple inheritance is not that language designers can't deal with these scenarios—either by introducing coping mechanisms that place the burden of resolution on the programmer, as in the ambiguity situation, or, as in the case of the circularity scenario, disallowing them completely. Rather, the problem is for the programmer. If a class is descended from three ancestors and each of those ancestors is descended from three others, etc., and one of these problems somehow creeps into the inheritance structure, the difficulty of isolating the source of the problem becomes exponentially harder with each level of inheritance. If your code is poorly documented, you're in trouble.

Opinion among language experts is by no means unanimous as to whether the advantages of multiple inheritance outweigh its disadvantages. Many object-oriented languages do without it, and indeed, until 1988, even C++ did not support it. Java does not support multiple inheritance, but instead introduces the concept of interfaces.

Interfaces provide a solution for many of the cases in which multiple inheritance is desirable, while avoiding its pitfalls. Because interfaces do not supply implementations for the methods they declare, there is no ambiguity in resolving where the definitions for those methods come from. Because all variables defined in an interface are both static and final, effectively serving as constants, they too are unambiguous. Finally, because of the distinction between classes and interfaces—in a sense, two separate inheritance chains, each with appropriate limitations—hidden circularities are avoided: there is no way that a class could find itself a hidden ancestor of any of the interfaces it implements.

How Interfaces are Used in Java

A classic example of the type of situation in which interfaces are useful is provided by examining, briefly, the way in which Java implements threads. Recall that a Java applet extends java.applet.Applet. However, it is very often the case that you want your applet to run as a thread, allowing other threads to execute simultaneously so as to more efficiently use system resources. However, if your applet extends java.applet.Applet, it cannot also extend java.lang.Thread, because that would violate Java's restriction against multiple inheritance.

To solve this problem, the designers of Java provided an interface, called java.lang.Runnable. If your

applet implements Runnable, you must define a method called run() and place the main code for your applet within this method. You then construct a Thread within your applet, passing this as a parameter to the Thread() constructor. The constructor for a Thread can take as a parameter any object that implements the Runnable interface. This constructor is defined as follows:

```
public Thread(Runnable target);
```

This constructor method constructs a Thread whose body, when it runs, is the run()method you provided in your applet. Notice that the parameter list of this constructor accepts an object of type Runnable. Here is the key to understanding how interfaces are used: If your applet implements Runnable, then your applet qualifies as an object of type Runnable. Any method, such as the constructor described earlier, that takes a Runnable as a parameter can accept your applet; when it calls the method(s) that are part of the Runnable interface—in this case only one method, run()—your applet has provided a method consistent with that request. It is as if there were an ordinary class called Runnable and your applet were derived from that class.

The Runnable interface is an excellent example for someone trying to understand interfaces for the first time, not only because it demonstrates a practical use for interfaces, but also because it illustrates an important concept in interface design, in fact an important principle in all programming, namely: keep it simple. There is only one method declared in the Runnable interface; thus, it is very easy to implement that interface in any class you design. If an interface were to define 20 methods, it's unlikely that anyone would consider it convenient to implement that interface. On the other hand, if those 20 methods were spread out among several different simpler interfaces, it would be no problem for the programmer to implement just those interfaces whose functionality was really necessary. Remember, you can implement as many or as few interfaces as you please.

Parts of an Interface Declaration

The general form for an Interface declaration is:

```
[interfaceModifiers] Interface InterfaceName
      [extends Interface1, ... InterfaceN] {
  variable declarations
  abstract method declarations
}
```

InterfaceModifiers

If declared without any modifiers, an interface is by default accessible only within the package. Otherwise, it can be public or abstract.

public

An interface declared as public is accessible to all classes everywhere, both inside and outside the package.

abstract

It is redundant to declare an interface as abstract, because all interfaces are inherently abstract. It is not, however, an error to explicitly declare an interface as abstract.

extends [Interface1, ... InterfaceN]

As we have pointed out, unlike a class, which can only extend one other class, an interface can extend as many interfaces as the programmer wants.

Summary

We've covered quite a bit of ground in this chapter. We have focused on bridging the gap between the abstract and the practical—taking the knowledge you gained in Chapter 1 about object-oriented programming concepts in general and bringing that knowledge to a practical examination of the specific ways in which these concepts are integrated into the structure of Java as a language.

We've seen how Java applies the notion of encapsulation, grouping data and methods together into classes. Also in the service of encapsulation, Java provides access specifiers (see Table 8.1), which allow the programmer to specify several levels of data hiding. Objects thus see as little as possible of the internal implementations of other objects, calling the other objects' own methods to operate on their instance variables and passing parameters as necessary.

In Table 8.1, under Variable and Method, an assumption is made to save space and preserve legibility. The conditions described for outside the package only hold true if the class is public. If the class is not public, then no variables or methods in that class will be visible outside the package, regardless of their access specifiers.

Table 8.1 Access Specifiers

	Class	Variable	Method
public	•Accessible to all classes, in and out of the package	•Visible to all classes with access to this class, in and out of the package •Visible in all classes derived from this one, in and out of the package	•Visible to all classes with access to this class, in and out of the package •Visible in all classes derived from this one, in and out of the package •Can be overridden in all classes derived from this one, in and out of the package
protected	Not applicable	•Inside the package:visible to all classes •Outside the package:visible only to classes derived from this one	•Inside the package: visible to all classes and can be overridden by classes derived from this one •Outside the package: visible only to classes derived from this one and can be overriden in these classes
[default]	•Inside the package:	•Inside the package: visible	•Inside the package: visible to all classes

	accessible to all classes •Outside the package: inaccessible	to all classes •Outside the package: visible only to classes derived from this one	and able to be overridden by classes derived from this one •Outside the package: visible only to classes derived from this one and can be overridden in these classes
private protected	Not applicable	•Visible only to classes derived from this one, both inside and outside the package	•Visible only to classes derived from this one, both inside and outside the package •May be overridden in derived classes
private	Not applicable	•Visible only within this class	•Visible only within this class
final	•May not be extended	•Assigned a permanent value in its declaration •Value may not be changed in any class, whether derived from this one or not	•May not be overridden
static	Not applicable	•One copy of this variable is shared by all subclasses	•This method is shared by all subclasses •May not be overridden
abstract	•Contains one or more abstract methods •May not be instantiated	Not applicable	•No definition (deferred to subclasses) •Must be overridden when subclassed, or subclass is also abstract
native	Not applicable	Not applicable	•This method is defined externally in another language (such as C)
synchron-ized	Not applicable	Not aplicable	•At a given time, only one thread can tell this instantiated object to execute this method •If static, only one thread can execute this method at a given time, period

Java's inheritance mechanism is simple but flexible, again using access specifiers so that inheritance can be controlled and shaped to the application designer's needs. Java avoids the complexity introduced by multiple inheritance, in favor of a simpler, less error-prone interface mechanism, which can be applied to many cases where multiple inheritance might have been used in languages such as C++.

Java supports both pure polymorphism (through the use of abstract classes and methods) and ad hoc polymorphism (through the use of overriding and overloading of methods).

At the same time, we have concentrated on the specific syntax for declaring classes, variables, methods, and interfaces. With these fundamentals firmly in place, we are now ready to move on and apply this knowledge, writing true object-oriented software in Java.

Java Programming Basics, by Mark Mangan, Edith Au, Jonathan Wallace, Ellick Quach, and Michael Wei.